# C++ pour la robotique

## C++ - Compléments de cours

Sébastien Rothhut

# Bibliothèque standard C++

Inclusion de fonctions et classes utilitaires

Exemples :

```
#include <iostream>
#include <cstring>
```

# Bibliothèque standard C++

Utilisation via le namespace std

Exemples :

```
using namespace std;
cout << “message”;
```

Ou :

```
std::cout << “message”;
```

# Fonctions utilitaires

iostream : flux d’entrée/sortie

- cin
- cout
- endl
- opérateurs << et >>

**Référence** : https://cplusplus.com/reference/iostream/

# Classes utilitaires

string : chaînes de caractères en tant qu'objet

- classe string
- utilitaires de conversion vers des types numériques : stoi, stol, stof, ...
- utilitaires de conversion depuis un type numérique : to_string

**Référence** : https://cplusplus.com/reference/string/

# Fonctions utilitaires issues du C

cstdio : gestion des entrées/sorties “à la mode C”

Lecture / écriture dans des fichiers, flux, strings, entrées/sorties standard

**Référence** : https://cplusplus.com/reference/cstdio/

# Fonctions utilitaires issues du C

cmath : fonctions mathématiques

- trigonométriques : sin, cos, tan, ...
- fonctions : exp, log, pow, sqrt, ...
- arrondis : ceil, floor, round, ...

**Référence** : https://cplusplus.com/reference/cmath/

# Fonctions utilitaires issues du C

cstring : utilitaires pour gérer des chaînes de caractères sous forme de tableaux de char

- concaténation
- comparaison
- recherche

**Référence** : https://cplusplus.com/reference/cstring/

# Fonctions utilitaires issues du C

Les fichiers d'en-tête préfixés par 'c' (par exemple cstring, cstdio, cstdlib) ont une appellation alternative issue du langage C (par exemple string.h, stdio.h, stdlib.h).

Les lignes `#include <cstring>` et `#include <string.h>` permettront d'utiliser les mêmes fonctions.

Cependant les fichiers *xxx*.h sont dépréciés. On préférera donc utiliser les fichiers c*xxx*.

# Fonctions utilitaires issues du C

Attention à ne pas confondre

`#include <string>` qui inclut la classe string

et

`#include <string.h>` qui inclut les fonctions utilitaires permettant de manipuler les tableaux de caractères se terminant par ‘\0’

# Directives du préprocesseur

Le préprocesseur fait une passe sur l'ensemble des fichiers du programme avant la compilation, et permet de faire des remplacements de caractères, de lignes ou même de décider s'il faut compiler ou non des portions entières de code.

On manipule son comportement via des directives, préfixées par '#'

# Directives du préprocesseur

```
#include <fichier d'en tête>
```

Permet d'inclure un fichier de la bibliothèque standard (localisation du fichier connue du compilateur)

```
#include "fichier d'en tête"
```

Permet d'inclure un fichier créé dans le cadre de votre programme (chemin absolu ou fichier présent dans le même répertoire)

# Directives du préprocesseur

```
#define SYMBOLE
```

Définit un symbole dont l'existence pourra être testée par la suite

```
#ifdef SYMBOLE
<lignes de code>
#endif
```

Si SYMBOLE a été défini, les lignes entre le #ifdef et le #endif seront compilées, sinon elles seront ignorées

# Directives du préprocesseur

```
#ifndef SYMBOLE
<lignes de code>
#endif
```

Instruction inverse : les lignes entre le #ifndef et le #endif seront compilées seulement si SYMBOLE n’a pas été défini

# Directives du préprocesseur

Cas pratique : éviter la double inclusion de fichiers d'en-tête

```
#ifndef _MON_FICHIER_
#define _MON_FICHIER_
<contenu du fichier MonFichier.h>
#endif
```

# Directives du préprocesseur

Cas pratique : éviter la double inclusion de fichiers d'en-tête

```
#ifndef _MON_FICHIER_
#define _MON_FICHIER_
<contenu du fichier MonFichier.h>
#endif
```

← Si **_MON_FICHIER_** n'a pas encore été défini

← Alors on définit le symbole **_MON_FICHIER_**

← Et on compile les déclarations du fichier d'en-tête

Si le fichier MonFichier.h est inclus plus d'une fois, seule la première occurrence sera compilée, les autres seront ignorées

# Directives du préprocesseur

`#define SYMBOLE <reste de la ligne>`

Toutes les occurrences de SYMBOLE dans le code seront remplacées par le reste de la ligne

**Exemple** : dans du code pour Arduino, on peut s'en servir pour renommer un pin particulier

```
#define PIN_SIGNAL_CAPTEUR 8
```

Toute référence à PIN_SIGNAL_CAPTEUR dans le code sera remplacée par la valeur 8

Cependant il est plus propre d'utiliser des constantes :

```
const int PIN_SIGNAL_CAPTEUR = 8;
```

# Rappels sur la lisibilité

Votre code doit être lisible par vous, votre binôme, le prof qui corrigera vos TDs...

Rappels :

- Indentation propre et cohérente
- Noms de variables, de fonctions, de classes explicites
- Commentaires utiles (dites ce que vous voulez faire, pas ce que vous faites)

```
int v = 42; // Initialisation de v à 42
```

```
int valeurInitCapteur = 42; // On prend une valeur de 42 car (...)
```

# Testez votre code

- Compilez et exécutez votre code fréquemment, pour repérer les erreurs le plus tôt possible
- Si vous rencontrez un problème, isolez la portion de code fautive :
  - instructions de debug (cout ou Serial.println() de valeurs intermédiaires)
  - vous pouvez écrire du code spécifique qui appellera une fonction isolément pour tester son comportement
  - utilisation d'un débogueur (ex: GDB) - hors programme du cours
- Testez les cas aux limites : si un paramètre est dépendant d'un capteur externe, comment se comportera votre code avec des valeurs imprévues ?

https://www.letemps.ch/sciences/une-incroyable-erreur-calcul-lorigine-crash-mars-module-europeen-schiaparelli